ARPA ORDER NO. 2223

12

ISI/RR-77-64
October 1977

Neil Goldman
Robert Balzer
David Wile

The Inference of Domain Structure from Informal Process Descriptions

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
ISI
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
14 ISI/RR-77-64

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
6 The Inference of Domain Structure from Informal Process Descriptions.

5. TYPE OF REPORT & PERIOD COVERED
9 Research rept.,

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
10 Neil/Goldman, Robert/Balzer, David/Wile

8. CONTRACT OR GRANT NUMBER(s)
15 DAHC 15-72-C-Ø308,

9. PERFORMING ORGANIZATION NAME AND ADDRESS
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
ARPA Order-2223

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE
11 Oct[redacted]977

13. NUMBER OF PAGES
25

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
12 26p.

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This paper was presented at the Workshop on Pattern-Directed Inference Systems, Hawaii, May 1977, and was published in the SIGART Newsletter 63, June 1977.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
artificial intelligence, domain models, inductive inference, natural language understanding, process specification, production systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Understanding informal descriptions of processes requires access to a body of knowledge about the process domain, and the ability to use that knowledge appropriately. A great deal of effort has been spent in developing methods for organizing and using domain knowledge; relatively little has been done to automate acquisition of such knowledge. Since English process descriptions
(continued)

→ next page

407 952

mt

20. (continued)

reflect the underlying structure of the process domain, knowledge about that structure may be inferred from the description itself. A categorization of important structural knowledge classes is presented, and a production system described which interprets English-like statements on the basis of existing structural context. A sample of the rules from this system is examined. By assuming conditions required in the rule patterns when a linguistic structure is not interpretable, it is possible to infer a great deal of structural knowledge about a process domain. This incremental growth of domain structure presents an alternative to constructing process understanding systems applicable only to very restricted domains, or requiring extensive additions of domain-specific knowledge by human experts for each new task.

ARPA ORDER NO. 2223

ISI/RR-77-64
October 1977

Neil Goldman
Robert Balzer
David Wile

The Inference of Domain Structure from Informal Process Descriptions

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA ISI 4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223.

CONTENTS

ABSTRACT

Understanding informal descriptions of processes requires access to a body of knowledge about the process domain, and the ability to use that knowledge appropriately. A great deal of effort has been spent in developing methods for organizing and using domain knowledge; relatively little has been done to automate acquisition of such knowledge.

Since English process descriptions reflect the underlying structure of the process domain, knowledge about that structure may be inferred from the description itself. A categorization of important structural knowledge classes is presented, and a production system described which interprets English-like statements on the basis of existing structural context. A sample of the rules from this system is examined. By assuming conditions required in the rule patterns when a linguistic structure is not interpretable, it is possible to infer a great deal of structural knowledge about a process domain. This incremental growth of domain structure presents an alternative to constructing process-understanding systems applicable only to very restricted domains, or requiring extensive additions of domain-specific knowledge by human experts for each new task.

1. INTRODUCTION

People can acquire an understanding of a process from a variety of sources -- for example, from repeated execution of the process under another person's guidance or from observation of another person carrying it out. Understanding is then a result of *generalization from experience*. An understanding of a process may also be acquired from a description, encoded in spoken or written English (the back of a parking ticket), in pictures (the back of a box of Minute Rice), or in a formal description language (the "algorithms" section of *Communications of the ACM*).

Process understanding can be measured along several dimensions. One measure is the ability to execute the process on actual "data," or to simulate it on "symbolic data." Ability to describe the process is another measure. Understanding may also be measured by the ability to prove theorems about the process, to give a rationale for its organization, or to modify it to meet altered goals.

The *SAFE* (Specification Acquisition from Experts) project at ISI has been investigating process understanding by translating a process description written in an informal, imprecise language with English-like semantics (hereinafter referred to as English) into a process specification language with formal syntax and semantics. A special interpreter exists for the latter language, so processes specified in its notation may be executed on appropriately encoded data.

The nature of English makes it unreasonable to define any mathematical measure of how well a process description is understood; this is solely within the competence of *human judges*--*in particular*, the human(s) who produced the English description. The formal specification itself, as well as observable behavior produced by its execution, may be used to make this judgment.

We have built an operating prototype process-understanding system, *SAFE*, which has "understood" three short (under 200 words) process descriptions. In this report no attempt is made to describe this entire system [3]. We confine our attention to linguistic aspects of the understanding problem, focusing on the use of a process domain model to interpret English statements.

Although some high level correspondence exists between our investigations and the issue of representation for episodic memory [1], our concern is not with the human ability to store process representations and execute them; rather, what we are attempting to automate is the ability to translate the ability to translate from one observable representation (English) into another observable representation (a formal operational specification) which we are attempting to automate. This translation is a central part of

This report presented as a paper at the Workshop on Pattern-Directed Inference Systems, Hawaii, May 1977, and published in the SIGART Newsletter 63, June 1977.

computer programming; the motivation for automating the task has been outlined elsewhere [3].

2. DOMAIN KNOWLEDGE

A compiler understands a formal process specification, such as an Algol program, at the *level* of "ability to execute," which requires no model of the process domain's structure. If a statement *EMPLOYEE [N,3] <- EMPLOYEE [N,3] + 30* increases an employee's Social Security number by 30 when it was intended to give him a raise, the program will produce incorrect results, but we do not blame this on the compiler. Nor should we, for the compiler has insufficient information to see anything wrong with this.

If the compiler is a human, however, and the specification is informal, more *intelligent* behavior is expected. If I ask the computer operator to "Logout Goldman" I do not expect as a reply "Goldman is not a job number". Rather, I expect the operator to convert the user name "Goldman" to the job number associated with that name, and logout that job number. Only if that conversion is ill-defined would I expect problems. If I ask someone to "delete my .TMP files" I expect him to fill in the implicit relation and delete files whose extension is ".TMP".

English descriptions of processes *are informal* in a variety of ways. Consider some examples from the *TENEX Executive Manual* [14]:

> *LOGOUT* . . . clears the user's job and returns it to the available job pool.

To understand this, we need to know (or infer) that an association can exist between a "user" and a "job", whence "clear" can indicate breaking that association. It also helps to view "available job pool" as a set of "jobs", whence "return" indicates an addition to that set, and "it" can sensibly refer to the "job" which was cleared.

> *TYPE* . . . To print symbolic files on your terminal . . .

Since terminals are output devices, it makes sense to print information on them. However, in a clause like "to list the names of files on directory SUBSYS . . .", the phrase "on directory SUBSYS" makes more sense when treated as a refinement to "files" than as the target location of "list".

> *LINK* . . . causes each of two users to be able to see output which is being typed on the other's terminal.

If our model of the underlying domain indicates that each user is associated with a particular terminal, this makes sense. But, if we have a more accurate model of TENEX, and know that terminals are associated with attached jobs, in a one-to-one fashion, and that jobs are associated with users, possibly many-to-one, then the "terminal for a user" is not well defined unless the user happens to have exactly one attached job. (In fact, the document proceeds to explain the action taken by LINK in the ill-defined cases.)

These examples demonstrate some of the ways in which the structure of a process domain can affect the interpretation placed on English descriptions. In this paper we shall consider domain knowledge only as it applies to interpreting semantic relationships between connected, intersentence linguistic constructs. However, the same knowledge must be used to analyze relationships between linguistically unconnected information in English process descriptions [3].

3. *REPRESENTING A PROCESS*

Many abstractions are available for specifying processes (e.g., Turing machines and Algol programs). While formally equivalent, the various representations have proved useful for different purposes. We have chosen an abstraction designed to limit the task of understanding English descriptions to one of removing informality, which consists primarily of the following:

- Resolving ambiguities.
- Filling in unspecified, but required, information.
- Explicitly linking information distributed in the description.

We view a process as the controlled application of *ACTIONs* to *OBJECTs*. The effect of applying an *ACTION* to *OBJECTs* may be to directly invoke further *ACTIONs*, to create new *OBJECTs* or destroy existing ones, and to create or destroy *ASSOCIATIONs* between *OBJECTs*. The environment in which the process operates consists of a data base of these associations, some of which may exist prior to its initiation. The control of the process consists of conventional programming language control structures: sequential invocation of actions, conditional invocation of actions based on the content of the data base, demonic invocation of actions based on additions to the data base, and iteration over sets of objects.

Objects, associations, and actions are primitive in this view. Issues of object and relationship representation are not addressable in this formalism, for we believe such issues rightfully belong in the task of process implementation, not process specification.

It was shown above how interpretation of informal statements about a process depends on a context of various types of objects interacting in particular ways. We call this context a domain model. In this report we are primarily concerned with the use of this model in understanding process descriptions and with techniques for inferring pieces of the model from the description itself. We begin by examining the components of a domain model.

4. *COMPONENTS OF DOMAIN MODELS*

An essential part of a domain's structure is the categorization of objects into *types*. TENEX, for example, manipulates jobs, users, files, I/O devices, directories, and many other object types. Virtually all the information we use about a domain is parameterized by the domain types.

The objects manipulated by a process are *instances* of these types. Most objects manipulated by a process are data to that process, that is, the objects themselves are not mentioned in a specification of the process. However, domains may contain distinguished objects which are explicitly mentioned in process descriptions. In the TENEX manual, for example, we find references to the I/O devices PTR and LPT, to the directory SUBSYS, and to access modes READ, WRITE, EXECUTE; these are all instances in the process domain.

Objects in the domain do not become associated in arbitrary combinations. TENEX maintains a table of associations between directories and passwords. LOGIN creates a new association between a job and a user. As a job accesses files, ternary associations between jobs, files and access modes are created. However, no meaningful associations exist between I/O devices and passwords, or between directories and dates. We use the relational data base formalism [7,9,11] to describe the permissible associations in the domain.

In this formalism, a *relation* is a time-varying set of *associations* (n-tuples). A given relation is defined over a fixed number of attributes. Each attribute of a relation is tied to a single type; several attributes may be tied to the same type. A tuple in a relation is a pairing of attributes with objects, subject to the restriction that an object may be paired with an attribute only if that object is an instance of the type to which the attribute is tied. We will define relations with the notation:

reldef (<relation> (<attribute> <type>). . .(<attribute> <type>))

For example, we might denote a relation control-tty by:

reldef (control-tty (tty terminal) (controllee job))

For tuples within a relation, we will use the notation:

<relation-name>((<attribute> <object>). . .(<attribute> <object>))

Thus, *control-tty ((tty TTY3) (controllee J7))* would indicate that the device TTY3 is associated with the job J7 in the control-tty relation. The ordering of attribute-type pairs in a relation definition, and of attribute-object pairs in a tuple, is arbitrary.

The *actions* performable in a domain cannot be applied to arbitrary operands, but only to instances of the appropriate type. TENEX can DELETE a file from a directory, LOGIN a user, LOGOUT a job, or ASSIGN a device to a job. But it makes no sense to LOGOUT a file or DETACH an account. We can capture the typing restrictions on operands to an action with a notation analogous to that used to define relations:

actdef (<action> (<attribute> <type>). . .(<attribute> <type>))

The types of a domain need not be disjoint. They fall naturally into a lattice under the relation *subtype*. "I/O device" may be subcategorized into "input device" and "output device," each a supertype of "terminal".

A domain generally has *constraints* on the state of the association data base. Certain combinations of tuples may not coexist, even though they are permissible individually. A common example of this situation is when a relation is constrained to be a function. We will indicate this by:

function (<relation> <attribute>)

The statement *function (control-tty tty)* would indicate that the control-tty relation defined above cannot simultaneously include two tuples with the same job but different controlling terminals.

Certain domain types have sets as instances. Such types are called *set-types*. "Directory" may be modeled as a set-type in the TENEX domain. Sets are generally homogeneous -- all members of a given set are instances of a single domain type. Furthermore, all sets which are instances of a given set-type will have the same typing restriction on their members. This restriction will be indicated by:

settype-elements (<set-type> <type>)

Thus, *settype-elements (directory file)* indicates that the members of any directory must be files.

The associations that exist at a given time during execution of a process need not be independent; the existence of one or more associations satisfying certain restrictions may imply the existence of other associations. If the implication holds at all times, the rule of implication is an *inference rule* of the domain and is part of the domain model. An example of such a rule would be, "If a file is named 'message,' then its non-owner protection is append-only."

In summary, a domain model consists of information about types of objects in the domain, the classes of associations which may be formed between those objects, the actions which may be performed on those objects, particular instances of the object types, particular associations which exist between those instances, etc. This information may be characterized as:

- time-independent -- Whereas any process creates and destroys associations (information) as it operates, the domain structure information remains static.

- constraining rather than determining -- Each piece of domain description serves to constrain the universe of processes which can be built within the domain. The domain structure does not determine a particular process, however, but only an infinite class of processes.

- needed for the non-performative aspects of process understanding rather than for actual process execution -- A process within the domain must conform to the constraints imposed by the domain, but has no need to access the descriptive information during its operation.

5. *SOURCES OF DOMAIN KNOWLEDGE*

Some knowledge of domain structure needed to understand an English process description may be "background" knowledge, not contained in the description itself. Construction and use of a large body of background knowledge has been a prime focus of much AI research, as it applies to process comprehension [16] and to more general natural language understanding as well [17].

Some experiments reported by Balzer [2], however, indicated that, for the degree of process understanding we are considering, humans are able to glean sufficient knowledge of domain structure from the text of the description itself. Such information is conveyed in two modes. One is the declaration--the explicit statement of structural information. Declarations are frequently available in formal languages. They are also commonly found in English process descriptions, and have been processed in various forms by a few English-like language processors [4,12].

Structural characteristics of the process domain may also be implicit in operational parts of an English description, playing a part in determining the words and syntactic structures chosen to describe the process. By judicious inference from observable English patterns and limited interaction with a knowledgeable human informant, it is possible to uncover much of the implicit structural information.

6. *LINGUISTIC ANALYSIS OF ENGLISH DESCRIPTIONS*

The first action taken by *SAFE* in understanding an English process description is to perform a linguistic analysis of the English, which serves two purposes. One is the conversion of the English to a "descriptive" format; the second is to detect new knowledge of the process's domain which is either implicit or explicit in the text.

The descriptive format is built from six description classes (some of which resemble the descriptor types of KRL [5]): event/relation descriptor (ED), object descriptor (OD), set descriptor, conditional descriptor, iteration descriptor, and conjunction descriptor. Only the first two of these will be of major concern in the remainder of this paper.

An ED is composed of an action or relation, and a (possibly empty) set of attribute-OD pairs, where the attributes are a subset of those in the definition of the specified action or relation. The ED thus describes an event or an association by naming the event's action or the association's relation and describing objects for some or all of the attributes of that action or relation. EDs will be indicated by:

[*<relation/action>* *<attribute>* = *<OD>* . . . *<attribute>* = *<OD>*]

ODs are composed of a domain type and any number of modifying EDs. An OD is a descriptive reference to a domain object; the referenced object must be an instance of the type indicated and must participate in events or associations as indicated by the EDs. ODs will be indicated by:

{ *<type>* *<ED>*. . .*<ED>* }

Each modifying ED must contain at least one attribute paired with the special symbol "*" which indicates the use of the described object in the modifying event/association.

Another form of reference is a direct pointer to a particular domain object. In all contexts in which an OD may be used as a reference, a direct pointer (internal object name) is acceptable.

In general, the linguistic analysis converts English noun phrases to ODs and clauses to EDs. For example, the clause "print the text" would become [*OUTPUT* ***info*** = {*TEXT*}] while "the printed text" would be converted to {*TEXT* [*OUTPUT* ***info*** = *] }

The linguistic analysis is performed by a production system. The rules in this production system have the general form

$$S(V),\ C(V) ==> A(V)$$

where S is a pattern (interpretation state) in terms of variables V which can match an

English structure, *C* is a condition based on domain structure, and *A* some action. When a structure being processed matches an S pattern, binding (some of) the variables *V*, and condition *C* holds, then action *A* is executed. This action is one which in general modifies the ED or OD being constructed as well as the English structure. The knowledge expressed by the rule is: "*A* is a procedural representation of the meaning of *S* under condition *C*".

In the "concrete noun" rule, for example, *S* matches a noun phrase with a concrete noun as its head, binding v_1 to that noun. *C* tests that v_1 names a known domain type, binding v_2 to that type. The action *A* then makes v_2 the type of the OD being constructed.

New structural knowledge of the domain can be obtained when an interpretation state S is matched but the corresponding condition C does not hold. By ***assuming*** C, the English can be understood. In the concrete noun rule, if the noun bound to v_1 was not known to name any domain type, a new domain type would be created and v_1 asserted to be the name of that type.

In summary, the production rules express the meaning of linguistic patterns, generally in terms conditional on the existing domain model. Some of the patterns are specific to particular English words or small classes of words; others are based on surface syntactic structures; still others are based on semantic patterns which could arise from *many different* words and syntactic forms. When the known structure is inadequate to make sense of the language, sufficient additional structure is *assumed* to enable application of a rule.

We now present a partial sample of the language analysis rules in *SAFE*. Nothing would be gained by an exhaustive listing, for the set of rules is not closed in any sense. Rules were generated as required by the example descriptions selected for the prototype system. Rules were added or generalized but not replaced for new examples. This sample is selected only from rules which can be used to infer structural properties of a domain.

The notions "modify" and "be an argument of" are commonly used in describing surface syntactic properties of English. In the discussion of the analysis rules, we extend these notions straightforwardly to the EDs and ODs which result from analyzing English structures. To present the rules precisely would necessitate introducing considerable notation. We shall simply describe the rules in terms of patterns, conditions, and actions and trust that the reader will see them as easily formalizable.

Verb Regularities

The actions and relations of a domain are generally named in English by verbs. In the surface syntactic structure of English, verbs are associated with cases, which are marked by word order or prepositions. When a verb is used to reference a particular action or relation, the cases of the verb and attributes of the action correspond in a predictable way. This case-attribute mapping for the verb "print" and the action "OUTPUT" is:

verbmap (print OUTPUT (object info) ((on at) where)) given:
actdef (OUTPUT (info file) (where output-device))

In other words, the direct object of the verb "print" can be expected to specify a file for the action OUTPUT, and the device which OUTPUT should write to is likely to occur following an "on" or an "at" preposition. Given a verb V, an action or relation R, and a surface case C, verbmaps determine a (possibly empty) set of attributes of R whose content could be encoded in case C of verb V. Similar uses of surface verb regularities can be found in a wide variety of natural language "parsers" [6,15]. The verbmaps enable *SAFE* to construct the argument pairs for EDs generated from English clauses.

Implicit Use of Domain Structure

Many rules are present to pick up references to domain structure which are implicit in the English -- that is, they are "presupposed" by the language used rather than being asserted by it. We shall categorize each rule roughly according to the category of domain structure with which it deals.

Type and Instance Rules. Use of concrete nouns in English is the most common method for naming types. When such a noun appears as the head of a noun phrase, it may name a domain type. "When a user types his password" would indicate that the nouns "user" and "password" name types.

Proper nouns, on the other hand, refer to particular objects. When a new proper noun (such as "SUBSYS") appears, it is assumed to name a new object. Since every object must be an instance of at least one domain type, a new type is created and kept on a list of "invented" types. In certain situations described below, an invented type may be replaced by another type, thereby revealing the new object as an instance of a known domain type.

Adjectives, like proper nouns, name particular objects. A new object and invented type are created when a new adjective is encountered. "Read-only" in "a read-only file", for example, names an instance of a type we might call "file protection".

When the surface argument of a verb is being processed, the verb's frame may predict a specific type for that argument. If the argument is transformed into an instance of an invented type, then the invented type may be merged with the expected one. For example, "print the file on the XGP" indicates that "XGP" is an instance of output-device, the type predicted by "print."

Relation Rules. When one OD modifies another, there is some relationship in the domain between their types. This situation can arise from a variety of English constructions, involving noun phrases, adjectives, proper nouns, and prepositional phrases (e.g., "a read-only file," "the LISP file," "the file protection," "the password for the user," "the user's teletype," etc.).

The implied relation may be direct as in "user's password," indirect as in "user's teletype" (short for "user's job's teletype") or ambiguous as in "user's files" (those in his directory? Those attached to his job?).

Certain verbs in English can relate two or more ODs and yield the same interpretation as the above constructs. Some examples are: "if the program *is* in LISP," "each user *has* a password," "the protection *assigned* to the file."

A variety of rules recognize this "description modifying description" situation. Among them are rules for recognizing NPs modified by NPs, NPs modified by adjectives, and NPs modified by prepositional phrases. All these rules convert the NPs involved to ODs, and build an intermediate form -- (MODIFIES OD_1 OD_2) -- eventually triggering another rule which encodes the knowledge for interpreting implied relationships (unless the ODs are in some way special and a more specific rule consumes the pattern first). When an implied relationship exists between two or more types, and the known domain structure contains no such relationship, this rule creates a new one. So "user's directory" could yield a new relation definition:

reldef (user-directory (u user) (dir directory))

Association Rules. When an OD modifies an instance, the OD can be assumed to describe the instance. This amounts to asserting the instance to be of the type present in the OD, and asserting each ED of the OD to hold on the instance. Thus, if "LISP file" were transformed into the OD {*file* [*source-language file* = ✳ *language* = *LISP*]} then "(the) LISP file MYPROG" would yield both the fact that "MYPROG" named an instance of a file, and that the source language of "MYPROG" was LISP.

Constraint Rules. The definite article "the" modifying an OD generally indicates that the description will reference a unique object in the context in which it is used -- perhaps simply because the process guarantees uniqueness in the particular context, or because the domain structure constrains the data so as to guarantee uniqueness in all contexts. In the latter situation the definite article may be "consumed" by an existing or assumed functionality constraint. So, even if it were not known that each TENEX user had a single password, an English phrase such as "the user's password" could yield the structural knowledge ***function (user-password pw).***

Explicit Declarations

A large number of linguistic patterns can indicate domain structure declarations. Consider the following:

STRUCTURE CATEGORY	EXAMPLE
Instance	<LISP> is a directory.
Constraint	Each directory has exactly one password.
Association	The password for <LISP> is LISP.
Constraint	The directory <DOCUMENTATION> may contain no read-protected files.
Subtype	All user-names must be directory names.

One feature common to these declarative forms is that the main verb does not name an action. In all cases, the resulting ED will contain a relation rather than an action. The distinctions between the various forms of declaration depend on whether arguments in this ED are ODs or instances, on universal and existential quantifiers, and on negatives. Notice that a declaration may have embedded within it constructions which rely on the implicit domain structure rules (see Constraint Rules) for their interpretation.

Other Rules

We mentioned that some rules are used to understand but not to infer domain structure. An example of this is a rule for nouns which reference relation attributes. The noun "number," for instance, is commonly used to refer to the "size" attribute of the cardinality relation, where

reldef (cardinality (s set) (size integer))

Such nouns are given case frames much like those of verbs. For "number" we have

nounmap (number cardinality (size)(of s))*

A rule exists which looks for noun phrases headed by common nouns with nounmaps defined on them. This rule enables *SAFE* to translate "the number of files" into an OD for an integer which is associated with a set of files under the relation "cardinality." We never make the inductive leap of inferring that a noun names an attribute of a relation; nounmaps exist only as built-in linguistic knowledge.

7. *EXAMPLE*

A short example will serve to demonstrate how the individual production rules work together in a way that permits simultaneous analysis of English statements and augmentation of the domain model. Suppose a domain model as depicted in Figure 1 exists. This model contains the facts:

- A directory is a set of files.
- A terminal is a kind of output device.
- A relation 'control-tty' associates jobs with terminals.
- A relation 'job-user' associates jobs with users.
- The action OUTPUT operates on a file and an output device.
- The action DELETE operates on a file.

We now examine the behavior of the production rules in understanding the following (artificial) example, which might be part of a description of the process of deleting a file:

(1) Print the filename of each file deleted on the user's terminal.

(2) Never delete the file in the directory of the user with filename 'MESSAGE'.

The actual input to SAFE is a parenthesized version of the English:

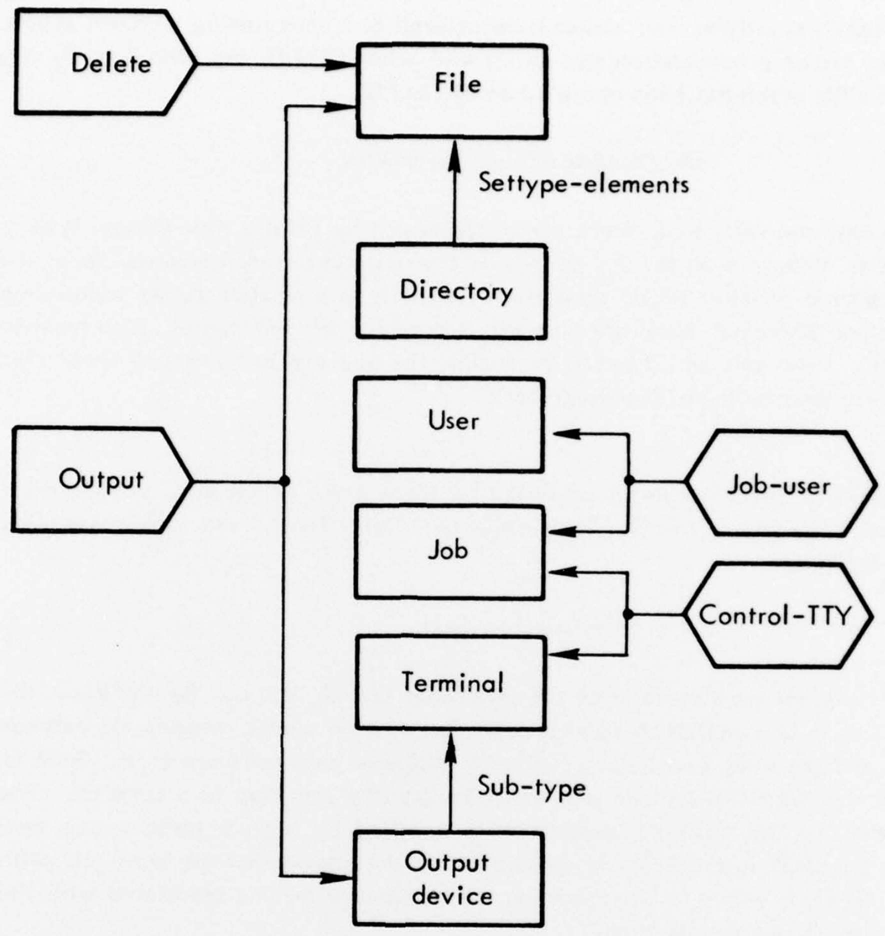
(1P) ((Print) (the filename of (each file ((deleted)))) on (the (user's) terminal))

(2P) (Never (delete) (the file in (the directory of (the user)) with ((filename) 'MESSAGE')))

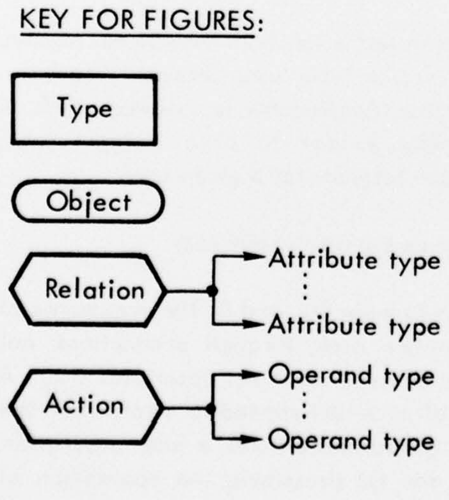
By parenthesizing clauses, verb phrases, and noun phrases at all levels, most common cases of syntactic ambiguity are removed. In (1P), for example, it is clear that ***on the user's terminal*** serves as an argument to ***print***, rather than an argument of ***delete*** or a post-modifier of ***name***.

. . . each file deleted . . .

The concrete noun (CN) production recognizes *file* as the name of a known domain type,



Delete
File
Settype-elements
Directory
User
Output
Job-user
Job
Control-TTY
Terminal
Sub-type
Output device



KEY FOR FIGURES:
Type
Object
Relation
Attribute type
Attribute type
Action
Operand type
Operand type

Figure 1

and builds an OD O_f specifying file. *Delete* is recognized as a verb naming a known action, and the relative clause production creates an ED with action DELETE and adds it to O_f. O_f now describes a file which has been operated on by DELETE.

. . . the filename of each file deleted . . .

Filename is a new concrete noun, which can be consumed by CN if a new domain type is created. This is done, and an OD O_{fn} specifying the new type is constructed. Now the prepositional phrase modifier (PPM) production can apply if a relation exists associating the types *file* and *filename*. None does, so one is created. We will call the new relation 'file-designator'. PPM adds an ED to O_{fn} restricting the filename to being one associated with the file described by O_f in 'file-designator'.

The definite article *the* can be consumed by the definite article (DA) production if 'file-designator' is in fact a function (one name per file). This is assumed, completing processing of the NP.

. . . the user's terminal . . .

Both *user* and *terminal* are consumed by CN, generating ODs O_u and O_t. Syntactically, the two nouns stand in a modification relationship. The domain model contains no relation between *user* and *terminal*, but it does contain a relational path between them. That is, 'user-job' can map from a user to a job, which 'control-tty' can map to a terminal. The domain condition on the MODIFIES production is satisfied by such a path, and a new relation, 'user-terminal' is created and defined to be the composition of 'user-job' with 'control-tty'. An ED is added to O_t restricting the terminal to be one associated with the user described by O_u in 'user-terminal'.

DA is once more applicable if 'user-terminal' is in fact a function. This is not known. (It would be if the two relations which were composed to form 'user-terminal' had been known to be functions.) To consume the definite article, functionality is assumed. This is actually an invalid assumption; since TENEX permits a user to have multiple jobs, 'user-terminal' does not necessarily determine a unique terminal for a given user.

Print the filename of each file deleted on the user's terminal.

The verb map for *print* causes the clause production to make O_{fn} and O_t the arguments of an ED with action OUTPUT. The universal quantifier *each*, through productions not described here, has embedded O_{fn} as the "generic element" of a set descriptor S_{fn}. A loop constructor production recognizes the pattern of an action applied to a set, when the actdef of the action does not specify a set. The production creates a loop descriptor, specifying a loop over S_{fn}. The loop's body is the ED describing the application of OUTPUT to O_{fn} (the set's generic element) and O_t.

In summary, linguistic analysis of this sentence has augmented the domain model by adding a new type, *filename*, a new relation, 'file-designator', and a relation 'user-terminal' defined as the composition of two previously known relations. P1 itself has been represented by a loop descriptor, saying in essence: "for each filename which is associated in 'file-designator' with any file which has been the operand of DELETE, output that filename on the terminal which is associated in 'control-tty' with a job associated in 'user-job' with the user."

Never delete the file in the directory of the user with filename 'MESSAGE'.

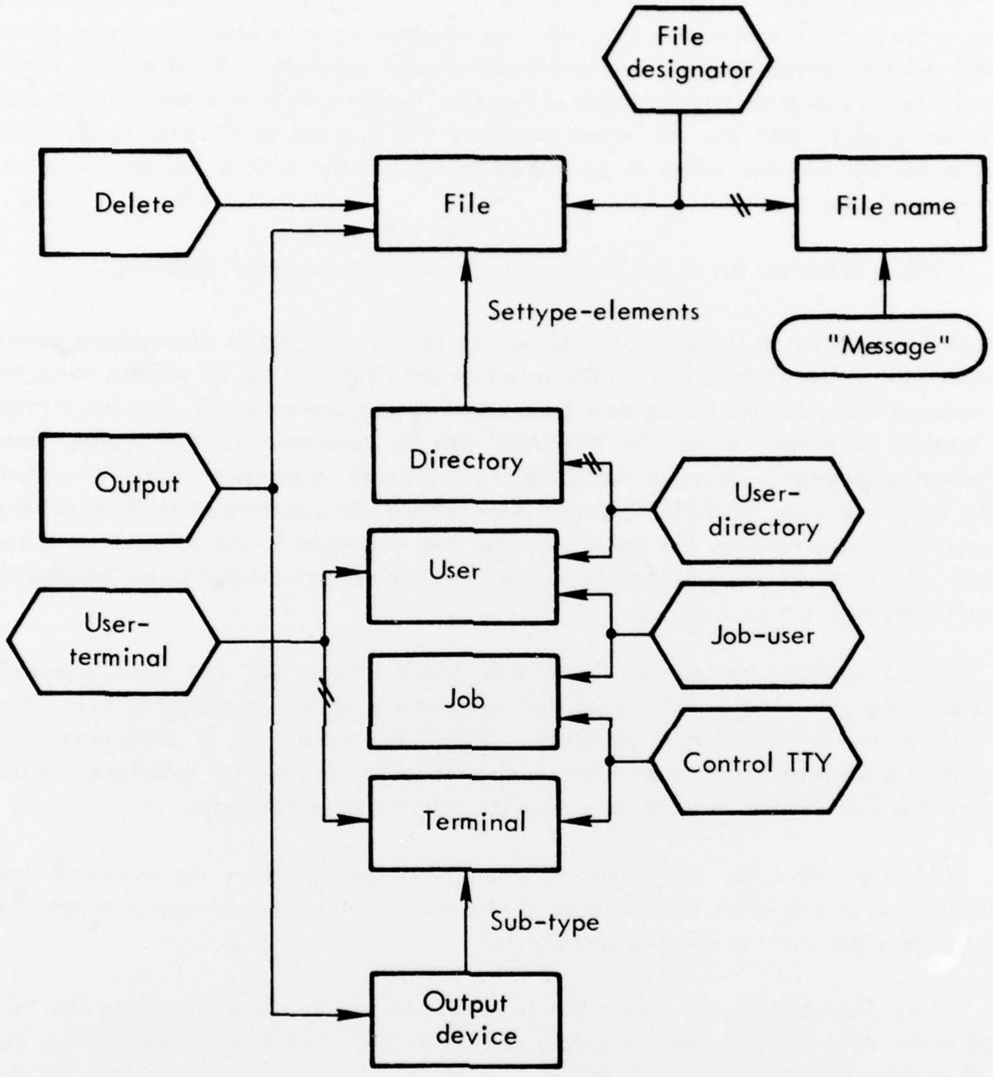
Our description of the analysis of (2) will be limited to highlights emphasizing points not observed in the analysis of (1). All the nouns are consumed by CN without need for any assumptions, *filename* having now been added to the domain model. Quoted strings are treated as proper nouns, so 'MESSAGE' can be consumed by the proper noun production providing it assumes 'MESSAGE' refers to an instance of some (invented) domain type. *Filename 'MESSAGE'* has now been refined into a pattern of an OD modifying an instance, which matches the pattern of the rule described in the Association Rules section. The rule can be applied by presuming that the object referred to by 'MESSAGE' is an instance of the type *filename*.

PPM explicates the implicit connection between *the file* and *with filename 'MESSAGE' via the relation* 'file-designator' created during the analysis of (1). The relational nature of the domain model thus allows the same rule to understand the selection of a filename based on a file description, as seen in (1), and the selection of a file based on the specification of a filename, as in (2), with the same data-path.

PPM explicates the connection between *directory* and *user* by assuming the existence of a new relation, 'user-directory'. DA assumes that this relation is a function (one directory per user) in order to consume *the*.

Finally, PPM handles the connection between *file* and *directory* by noting the fact stored in the domain model that a directory is a set of files. 'Set-membership' is thus the implicit relation between *file* and *directory*. The entire NP *the file in the directory of the user with filename 'MESSAGE'* has been converted to an OD specifying "an object of type file which (i) is a member of the directory associated with the user in 'user-directory', and (ii) is associated with the filename 'MESSAGE' in 'file-designator'.

Figure 2 depicts the state of the domain model following linguistic analysis of the pair of example sentences. The exact initial state of the domain model was not crucial to SAFE's ability to analyze the sentences. Nor was the precise wording of the sentences themselves crucial. A reasonable, but not isomorphic, analysis would result even if the initial domain model were empty and only the verb maps and syntactic knowledge available. A more complete initial domain model could reduce the number of assumptions needed to assimilate the sentences. The final domain model, of course, is sufficient to enable reanalysis of the sentences with no assumptions.



—⫽— indicates functionality of relation

Figure 2

8. *IMPLEMENTATION*

It would be possible to use these production rules as independent meaning derivation rules. If sufficient background knowledge of the process domain existed, this would probably provide an adequate linguistic analysis for our understanding task. Using the rules to acquire domain knowledge, however, introduces certain complications:

- An English structure may match several rule patterns but none of the corresponding conditions may be satisfied.
- It may be possible to satisfy a given condition with several alternative assumptions.
- It is desirable to try more specific rules with complex conditions before trying general rules. When all rules fail, however, the assumptions needed to satisfy the general rules are usually simpler than those for the complex rules, although the exact assumptions required depend on existing structural knowledge.

Greater control over the structural assumptions is required than is provided by a straightforward production system implementation. To obtain this control, we have chosen to use the rules directly only for analysis. Assumptions are handled by adding special rules containing English patterns without structural conditions. The action for such rules is simply a hand-coded function which decides on the "most reasonable" assumption to make, given the existing context. In all cases that assumption is one which would permit one of the production rules to work in analysis mode; thus, if the English phrase leading to the assumption should reappear, no new assumption.will be needed.

The availability of multiple satisfactory assumptions leads to a classic AI search problem. It is conceivable that we could carry multiple domain models through the linguistic analysis, choosing one on a simplicity basis in the end. Backup, however, does not seem to be an applicable approach. There are no "dead ends" in the analysis; it is always possible to make enough structural assumptions to continue. Currently, however, we have chosen to rely on human expertise to choose between alternatives. Whenever the analysis reaches a stage where alternative assumptions are acceptable, and frequently when only one is, the system asks for external verification. While we do not yet understand the tradeoffs between increased interaction and carrying uncertainty through the analysis, this verification of assumptions can be seen as a method used by humans in understanding process descriptions [2].

The rule patterns described in Section 6 are not directly applicable to English text; they are defined in terms of surface structure relations such as "prepositional phrase modifying noun phrase" or "adjective modifying noun." Recognition and representation of these surface relationships is done explicitly by some natural language systems [13], and implicitly by others. However, real English sentences cannot be unambiguously parsed on a purely syntactic basis. We believe that interpretation rules like those incorporated in *SAFE* will eventually be used in natural language understanding systems, since they are fundamental to disambiguation which cannot be dealt with by syntax alone.

Merging the production-based understander with a syntactic processor is not an immediate concern of *SAFE*, however. English is but one possible input language for a process understanding system, and is not likely to become a practical one for many years. An artificial language with English-like semantics is a more likely candidate at present. We are currently utilizing the parenthesized English input exemplified in the example. At the cost of having a human pre-processor make the necessary decisions about scope of conjunctions and dependency of modifiers, we are thus able to process a language which has no "syntactic" ambiguity but which retains most of the semantic properties of natural English.

9. CONCLUSIONS

It is well known that natural language understanding requires the use of domain-specific knowledge. When the language understanding task is restricted to that of making informal operational descriptions of processes precise, the breadth of such domain-specific knowledge required is reduced to the abstract domain model (Section 4). The full task of making informal descriptions precise requires additional, domain-independent, knowledge; in particular, it requires extensive knowledge of what makes a precise process description well-formed [3].

The words and structures chosen to describe a process in a given natural language reflect the regularities in the domain of that process [10]. In many cases, these words and structures can be used to infer the underlying regularities. To the extent this inference process can be automated, it will be possible for a machine to understand a process description without the need for a human to build a complete, precise, consistent description of the processing domain.

Automatic construction of domain descriptions is desirable for several reasons. Hand coding of the models is tedious and error-prone work; it is difficult for the same reasons that formal specification of processes is difficult. Domain description is a skill requiring special training, and is not an appropriate task for the user of the *SAFE* system.

Furthermore, a domain-specific system would confine the user to a predefined model. Besides lacking generality, this constraint may make it difficult for a user to determine if his application falls within the prescribed bounds, particularly if there exist alternative views of the domain. Manual construction of a domain model by a "human modeling expert," drawing inferences from the informal specifications as *SAFE* does, is reasonable only if that expert can be both thorough in seeing all alternative interpretations and meticulous in recording his assumptions. The former is probably impossible for any expert, and the latter extremely difficult. Both are properties of tasks better suited to mechanical execution.

We believe that the progress made on *SAFE* demonstrates the feasibility of automating the construction of domain models from informal specifications, a requisite part of a domain-independent specification understanding system.

REFERENCES

1. Abelson, R., "Concepts for representing mundane reality in plans," in D. Bobrow and A. Collins (eds.), *Representation and Understanding*, New York, Academic Press, 1975, pp. 273-309.

2. Balzer, R., *Human Use of World Knowledge*, USC/Information Sciences Institute, ISI-RR-73-7, March 1974.

3. Balzer, R., Neil Goldman and David Wile, "Informality in program specifications," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Boston, Mass., August 1977, pp. 389-397.

4. Bobrow, D., "Natural language input for a computer problem-solving system," in M. Minsky (ed.), *Semantic Information Processing*, MIT Press, Cambridge, Mass., 1968.

5. Bobrow, D. and T. Winograd, *An Overview of KRL, a Knowledge Representation Language*, Xerox Palo Alto Research Center, CSL-76-4, July 1976.

6. Celce-Murcia, M., "Verb paradigms for sentence recognition," *American Journal of Computational Linguistics*, 1976, 1, microfiche 38.

7. Chamberlin, D., "Relational data base management systems," *Computing Surveys*, 8, 1, 1976, pp. 43-66.

8. Chodorow, M., and L. Miller, *The Interpretation of Temporal Order in Coordinate Conjunction*, IBM RC-6199, September 1976.

9. Codd, E. F., "A relational model of data for large shared data banks," *Communications of the ACM*, 13,6, 1970, pp. 377-387.

10. Goldman, Neil, "Conceptual generation," in R. Schank (ed.) *Conceptual Information Processing*, American Elsevier, New York, 1975.

11. Grossman, R., *Some Data Base Applications of Constraint Expressions*, MIT Laboratory for Computer Science, TR-158, 1976.

12. Hayes, J. R. and H. Simon, "Understanding written problem instructions," in Gregg (ed.) *Knowledge and Cognition*, Lawrence Erlbaum Associates, Potomac, Md., 1974.

13. Kaplan, R., "A general syntactic processor in natural language processing," in R. Rustin (ed.), *Natural Language Processing*, Algorithmics Press, New York, 1973, pp. 193-241.

14. Myer, T., and J. Barnaby, *TENEX Executive Manual*, Bolt, Beranek and Newman, Inc., April 1973.

15. Riesbeck, C., and R. Schank, *Comprehension by Computer: Analysis of Sentences in Context*, Yale University, Dept. of Computer Science, Research Report 78, October 1976.

16. Ruth, G., *Protosystem I: An Automatic Programming System Prototype*, MIT Laboratory for Computer Science, TM-72, 1976.

17. Schank, R., and R. Abelson, "Scripts, plans, and knowledge," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, MIT Artificial Intelligence Laboratory, 1975, 151-157.